Behavioural Description Formalisms for Service Integration - Survey and Comparation

Technical Report No IC/2002/24
Date: 22 May 2002

Ion Constantinescu, Boi Faltings

Laboratoire d'Intelligence Artificielle, Department Informatique,
Swiss Federal Institute of Technology, IN (Ecublens), CH-1015 Lausanne, Switzerland.
{ion.constantinescu,boi.faltings}@epfl.ch

Abstract. Current web infrastructure is oriented on human-machine interactions. Developments for next generation large-scale information systems (such as Web Services and ebXML) however aim to allow for automated interactions between arbitrary systems / services. This is achieved by supplying high level descriptions of service capabilities and enabling information systems to dynamically discover and access each other.
Fundamental to progress towards this vision is the understanding of the behavioural descriptions that services might publish/access in such new information environments. The properties of such formalisms will have major impact on the practically of building systems able to effectively reason about service integration.
This paper reviews 5 well known behavioural description languages (PDDL, DAML-S, WSFL, ebXML and ConGolog) and makes an initial evaluation on the basis of a number of criteria such as completeness, flexibility, composability, expressiveness and complexity.

Keywords: service integration, interaction protocols, process definition, planning, distributed artificial intelligence, multiagent systems.

1 Introduction

Today, the largest human designed and controlled environment is the Internet. The Internet can be seen as a dynamic environment with a huge heterogeneous collection of infrastructures and services. There are many different types of architectures and severals metaphors for using them but central to all is the concept of "service" - providers of static (e.g. web pages) or active (e.g. pizza ordering) resources. Active resources allow on their invocation for changes in the environment to be effected. As the number of services increases so does the need for service reuse and service composability.

The underlying problem of service integration which needs to be solved to realise the above vision is directly related to much of the research that has been carried out in the Agent and AI communities. Service integration can be seen

as a complex coordination problem [19] (since services must work together over a period of time to achieve the initial user request) with a reasoning problem at its core (constructing a multi-agent plan to determine how services will work together).

One critical factor in building successful systems able to dynamically discover services in the environment and reason about how they might work together is the formalism used to describe services.[1] Service descriptions can be divided into two types:

- Structural descriptions - which define capabilities like data types and operation signatures and for that a number of well established standards like WSDL, IDL or RDF already exists.
- Behaviour descriptions - which deal with interaction patterns and process definitions.

In this paper we are concerned with the later. For that we review 5 formalisms (PDDL, DAML-S, ebXML, WSFL and ConGolog) which could be used for service descriptions. Our objectives in doing so are:

- To compare the properties of these (very different) formalisms.
- To provide a baseline for discussion on practical requirements for behavioural service descriptions in future on-line information systems.

This papers is organised as follows: in Section 2 we present a general model covering all aspects that we consider relevant for service integration. Section 4 represents the core of this work and lists a number of formalisms for defining behaviour including PDDL, DAML-S, ebXML, WSFL and ConGolog. Section 5 briefly describes usage examples. For a more in depth description of those formalisms see [5]. Section 6 defines a number of essential features for behavioural formalisms and tries to do a comparation between them. Finally Section 7 draws some preliminary conclusions and lists important discussion points.

2 General model

Our work relies on a general model containing mainly two steps: **definition** and **deployment**. The **definition** step is in close relation with the requirements and design of a service. The service definition is done in a formal way using structural and behavioural formalisms (see Figure 1). Structural descriptions define capabilities like data types and operation signatures. Behaviour definitions describe how existing or hypothetic services work or should work. The focus of this work is on formalisms for behaviour definition.

Deployment is the process by which a service gets actually used. Either from a service consumer perspective (e.g. user) which has some needs and requests some results or from a service producer perspective which wants to make available his capabilities for a given reason.

[1] In fact poor choices here are likely to give rise to intractable reasoning problems.

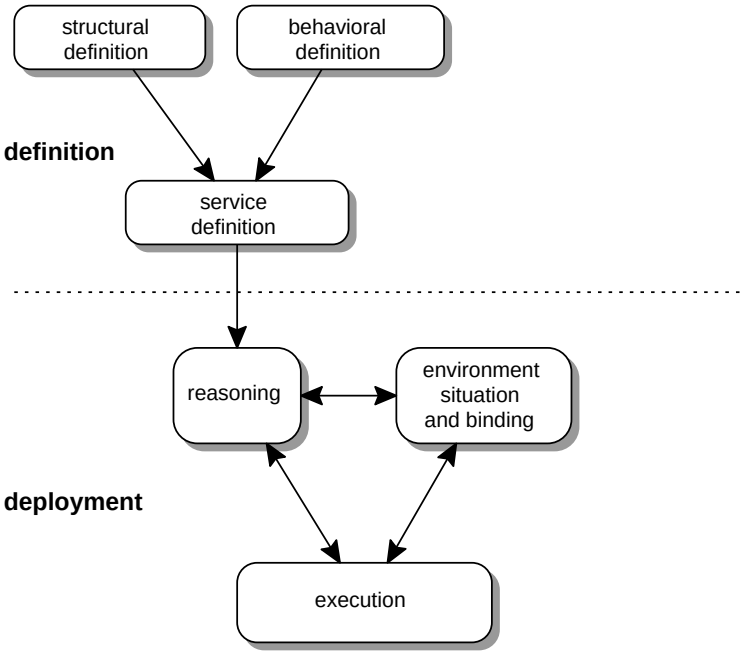

Fig. 1. General model for service definition and deployment

Figure 2 describes some of the interactions between the three elements of our system and exterior entities such as users, sensors, binding repositories/matchmaking systems and actual service instances.

3 Notations

Through the rest of this document we are going to rely heavily on the Unified Modelling Language (UML) notation for describing different formalism. Figure 3 shows the notation elements used in this document.

4 Existing formalism for behavioural definition

This document focuses on the expresivness of planning formalisms rather than exhaustively analysing all the aspects of the considered languages.

4.1 Planning Domain Definition Language

The Planning Domain Definition Language (PDDL) [10], [9], [8], was developed as a problem-specification language for the AIPS-98 planning competition.

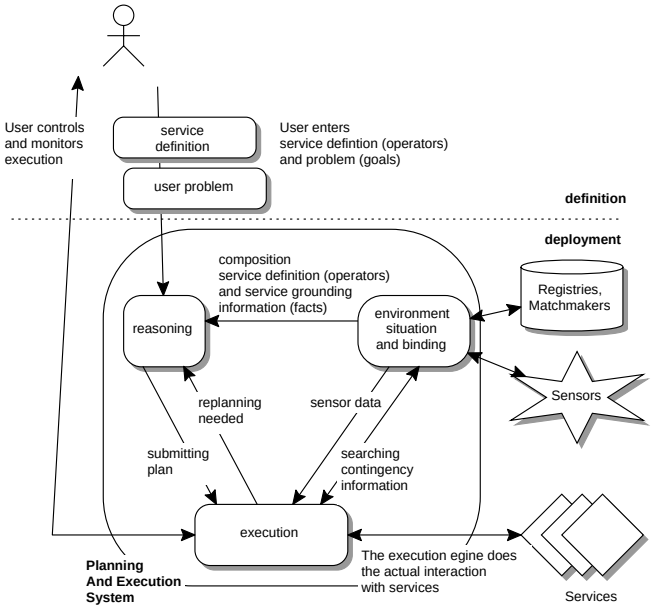

Fig. 2. Service Integration using a Planning-Execution System

PDDL draws from existing formalisms like ADL, SIPE-2, Prodigy-4.0, UMCP, Unpop and UCPOP.

Some of the syntactic features supported by the language are:

- Basic STRIPS-style actions
- Conditional effects
- Universal quantification over dynamic universes (e.g. object creation and destruction)
- Domain axioms over stratified theories
- Specification of safety constraints
- Specification of hierarchical actions composed of sub-actions and subgoals
- Management of multiple problems in multiple domains using differing subsets of language features (to support sharing of domains across different planners that handle varying levels of expressiveness)

The language is split into subsets of features, called *requirements*. Domain definitions declare which requirements they assume. The language is also structured in 5 levels, each level having specific *requirements*.

The language is defined by several documents. The first defines PDDL version 1.0 which was used in the AIPS 98 competition and corresponds to level 1 of PDDL2.1. The second defines PDDL2.1, makes reference to 1.0 and defines

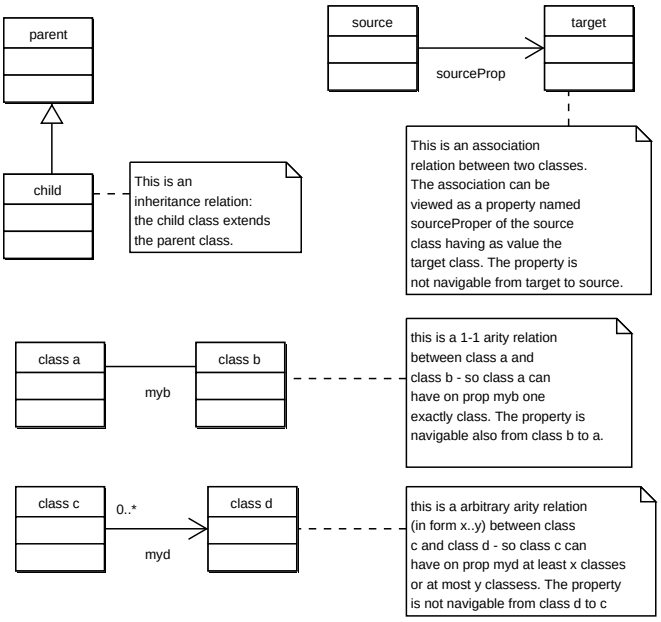

Fig. 3. UML Notations

levels 2 trough 4. PDDL2.1 is used by the AIPS 2002 competition. A third document defines level 5 named also PDDL+.

As mentioned before level 1 defines a basic propositional language and is specified by PDDL1.0.

Level 2 of PDDL provides support for numbers by allowing numeric quantities to be assigned and updated. These are captured as numeric fluents because they express change. Fluents, named also functional-expressions are not allowed to appear as arguments to predicates or to appear in terms.

Level 3 introduces the concept of time and durative actions in order to better model continuous processes as these arise in planning problems. As such a level 3 plan is a sequence of actions happening at defined time moments. It is possible to have multiple actions with the same time which indicates that they should be executed concurrently.

Still at this level there are no continuously changing quantities troghout the course of an action. All conditions and effects of durative actions must be temporally annotated. This is done by making explicit if an associated proposition must be asserted at the *start* of the action duration interval, at the *end* of the interval or is an invariant over the duration of the action. For level 3 no other

points are accessible, so all discrete activity takes place at the identified start and end points of the actions in the plan.

There is at least one possibility to convert level 3 plans to level 2. This can be done by introducing supplementary start/end actions to represent the end points of the duration interval. Invariants are converted by introducing checking actions between pairs of happenings in the original plan.

In level 4 numeric values can change continuously and are accessible at arbitrary points on the time-line off the plan. Continuous effects are not temporally annotated because they can be evaluated at any time during the interval of the action. At this level actions assigning, consulting and continuously modifying the same numeric variables can happen concurrently.

It can be argued that level 3 can express some of the plans in level 4 by having the domain designer anticipate every useful combinations of behaviours and ensure that appropriate encapsulations are provided.

Level 5 of the language introduces two new concepts besides actions: *processes* and *events*. Also the domain features that might be modelled as actions with durations are modelled as *start-process-stop* structures consisting of a point of initiation, a process and a point of termination. The points of initiation and termination can be application of actions, events or points at which the effects of active processes cause numeric values to reach critical thresholds.

In the next paragraphs we are going to describe levels 1-4 of the language. Level 5 is considered to not have yet a consistent body of work behind it so it is not taking into account for now.

As seen in Figure 4 the core concepts of PDDL are the *problem* and the *domain*.

A problem is what a planner tries to solve.

A PDDL problem is defined in respect to a domain by reffering to the domain's name.

Apart the domain a problem specifies also: an initial situation (*init-el*), and a goal to be achieved (*a-GD*).

The initial situation is described as a list of predicates that are to be considered true or value assignments for *fluents*.

The goal to be achieved is specified in terms of a simple (non-durative) goal definitions.

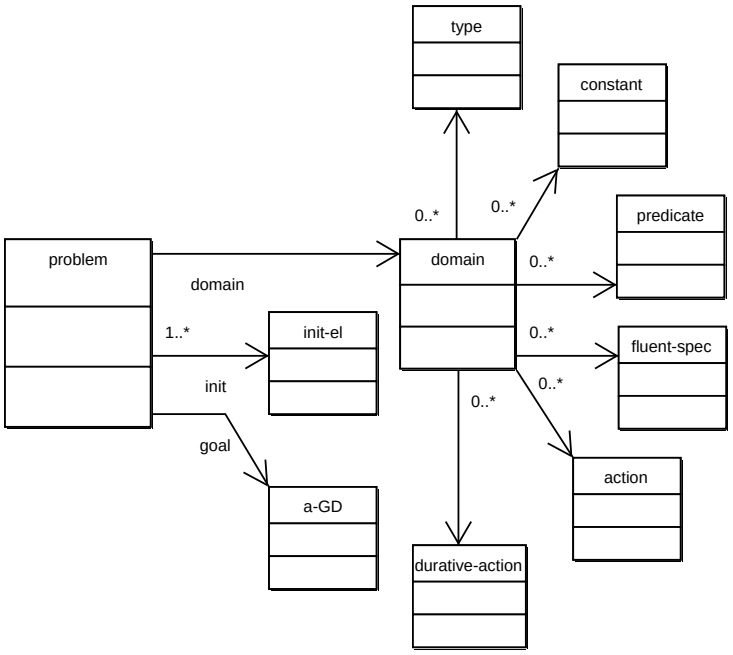

Fig. 4. PDDL - Problem Domain

A domain defines the following:

- zero or more data types
- zero or more constant values
- the predicates that are going to be used in action descriptions
- the numeric fluents that are going to be used in action descriptions (for level 2 and above)
- action definitions
- durative action definitions (for level 3 and above)

Actions (normal or durative) are the most important items defined by a domain. Figure 5 defines the relations between domains, normal actions and durative actions.

An action has zero or more *parameters*. Parameters are variables allowed to appear in the expressions defined by the action like precondition, effect and expansion actions.

An action has an optional *precondition* defined as a Goal Definition *a-GD* expression - see below Figure 6. This expression must be satisfied before the action is applied. If no precondition is specified then the action is always executable.

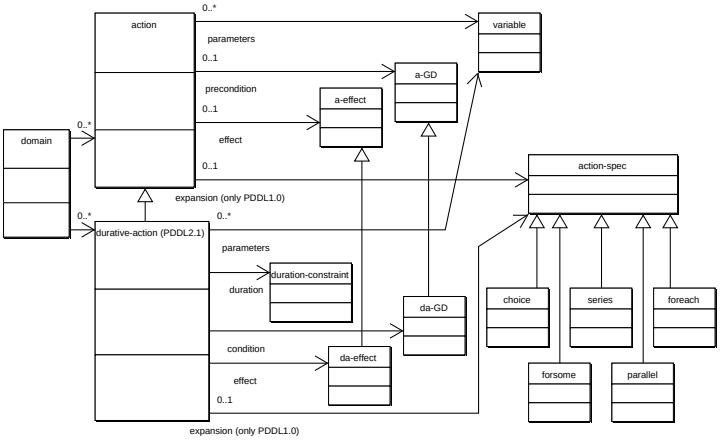

Fig. 5. PDDL - Action and Durative Action

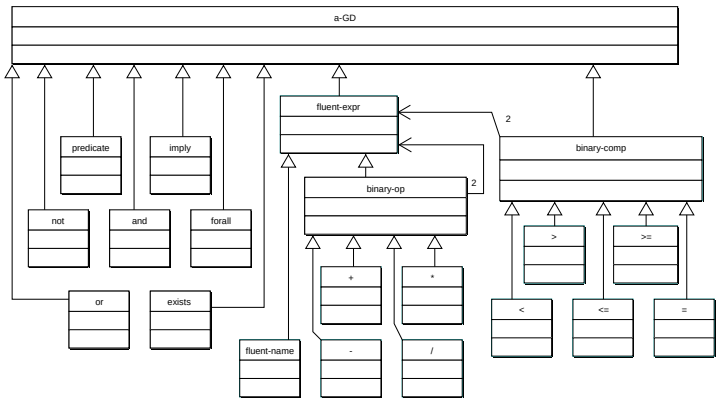

Fig. 6. PDDL - Action Goal Definition

Each action has an optional *effect* defined as an *a-effect* expression - see Figure 7. The *a-effect* expression defines the changes that the execution of the action has on the state of the world.

In PDDL 1.0 each action defined also an expansion slot allowing for definition of hierarchical actions in terms of *action-spec* expressions. Composition primitives like *choice*, *series*, *parallel*, *foreach*, *forsome*. In PDDL2.1 this slot was removed.

a-GD expressions (Figure 6) allow arbitrary first order logical sentences (and, or, not, exist, forall, imply) including expressions with operations (+, -, *, /) and comparations (<, >, <=, >=, =).

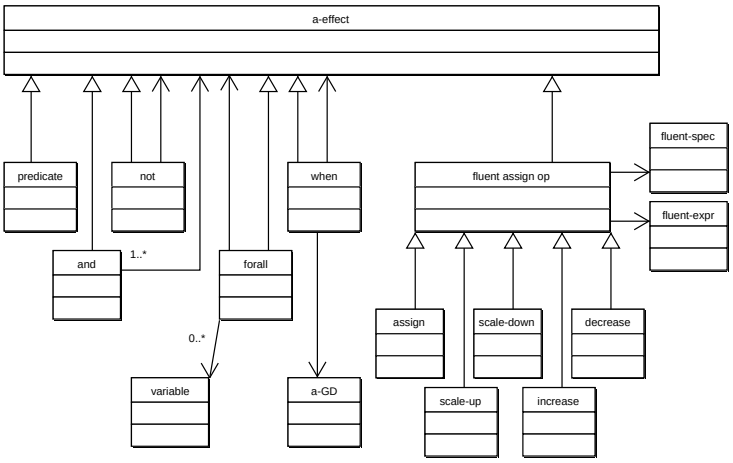

Fig. 7. PDDL - Action Effect Definition

a-effect expressions (Figure 7) allow for universal quantification (forall), conditional effects (when construct), fluents and fluents assignments operations (assign, scale-up, scale-down, increase, decrease). Full first order sentences (e.g. disjunction and Skolem functions) are not allowed. Also nested conditional effects are not allowed.

The definition of a durative action extends the definition of normal actions by allowing the specification of a *duration* and extended specifications for condition and effect (*da-GD* and respectively *da-effect*).

4.2 DAML Semantic Markup for Web Services - Process Ontology

DAML-S - stands for DARPA Agent Markup Language (DAML) Semantic Markup for Web Services (see [6], [2], [17], [18]). In the specification document services are defined as:

"sites that do not merely provide static information but, allow one to effect some action or change in the world"

DAML-S is targeted to enable the automation of the following tasks:

- Web service discovery - involves the location of Web services that provide a particular service and that adhere to requested constraints.
- Web service invocation - involves the automatic execution of an identified Web service by a computer program or agent.
- Web service composition and interoperation - involves the automatic selection, composition and interoperation of Web services to perform some task, given a high-level description of an objective.

– Web service execution monitoring - involves mechanisms and execution models that allow users or agents to determine the state of long-running services and interact with their execution

The current DAML-S specification (version 0.6) is defined by a number of documents: a technical overview document, a walk-through, a number of DAML-based ontologies and finally a number of example DAML-S services.

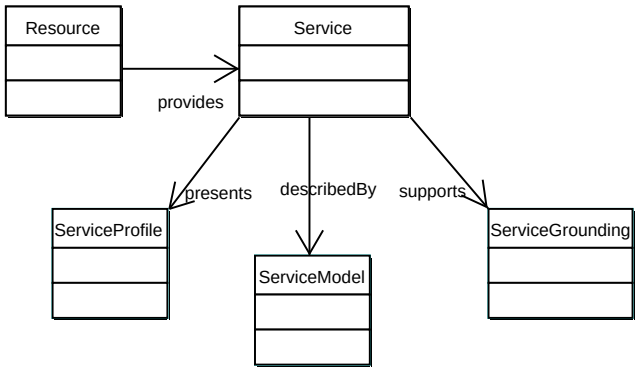

Fig. 8. DAML-S Ontologies: Service, ServiceProfile, ServiceModel and ServiceGrounding

DAML-S architecture comprises several ontologies (depicted in Figure 8) in the DAML+OIL (see [12], [7]) markup language:

– Service - an upper-level ontology defining an abstract concept of Service which is to be later extended. Presents, describedBy, and supports are properties of the abstract *Service* class. They can take as values *ServiceProfile*, *ServiceModel*, and respectively *ServiceGrounding* classes.
– Service Profile - responds to the question "what the service does ?"; that is, it gives the type of information needed by a service-seeking agent to determine whether the service meets its needs (typically such things as input and output types, preconditions and postconditions, and binding patterns).
– Service Model - responds to the question "how the service works ?"; that is, it describes what happens when the service is carried out.
– Service Grounding - responds to the question "how to access it ?"; that is it specifies the details of how an agent can access a service. Typically a grounding will specify a communication protocol (e.g., RPC, HTTP-FORM, CORBA IDL, SOAP, Java RMI), and service-specific details such as port numbers used in contacting the service.

In this section we are going to analyse and evaluate the Service Model, considering that Service Profiles address a different problematic. Service Groundings are not yet define and are used only as stubs.

The DAML-S specification chooses to model service behaviours as processes. For that the *ServiceModel* class reffered by the top-level *Service* class is extended as a *ProcessModel* class.

The two main components that form a *ProcessModel* are the *ProcessClass* and the *ProcessControlModel*.

A *Process* defines a service in terms of inputs, outputs, preconditions, effects and where appropriate it's component subprocesses allowing for higher level reasoning regarding service composition and interoperation. In DAML-S application processes are to be defined in terms of classes which get instantiated for each execution. As such the ProcessClass is the set of all subsets of Process. We approximated that with the upper class of all kind of *Process*es. All DAML-S Process related definitions are currently collected in the Process Ontology.

Execution control and monitoring of a service request is to be handled trough the *ProcessControlModel* - a top level top-level defined currenlty only as a placeholder. A ProcessControlOntology is planned for the future.

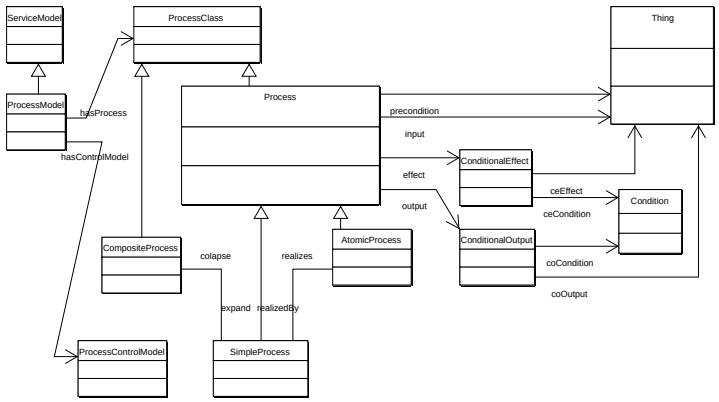

Fig. 9. DAML-S Process Definition

The Process Ontology (Figure 9) defines four main classes: *Process*, *SimpleProcess*, *AtomicProcess*, *CompositeProcess*. Definitions in the Process Ontology draw from the following areas:

- PDDL
- PSL
- Workflow Management Coalition Effort
- GoLog and ConGolog

The *Process* class is defined as the union of the three other classes. The *SimpleProcess* and *AtomicProcess* classes directly extend *Process*. This definition intends to capture the different nature of the *CompositeProcess* class.

The *SimpleProcess* class is introduced in order to provide a suplementar (and optional) level of abstraction. As such both *AtomicProcess*es and *CompositeProcess*es can be normalised to a *SimpleProcess* class. In that way we could imaging that computations for a given level of a problem could be done only in terms of *SimpleProcess*es.

AtomicProcess classes are the basic units of implementation. To interact with an atomic process involves (at most) 2 messages: one carrying its inputs, and one carrying its outputs. At deployment an *AtomicProcess* is required to be associated with a Grounding. An atomic process is a "black box" representation; that is, no description is given of how the process works (apart from inputs, outputs, preconditions, and effects).

*CompositeProcess*es are composed of subprocesses, and specify constraints on the ordering and conditional execution of these subprocesses. *CompositeProcess*es bottom out in non-composite (atomic and/or simple) processes.

The relations between a *SimpleProcess* class and *AtomicProcess* and *CompositeProcess* classes are bijective. As such a simple process that abstracts an atomic process is "realized by" that process. The other way around an *AtomicProcess* "realizes" a *SimpleProcess*. For the *CompositeProcess* the "expand" of a *SimpleProcess* to a *CompositeProcess* can be seen as a zoom-in and the inverse "collapse" coresponds to a zoom-out).

The main properties of a process are his *input*s, *output*s, *precondition*s and *effect*s. For processes a generic property named *parameter* is defined which is then extended by *input*s and *output*s. *input*s and *precondition*s can contain any kind of definition expressed as the DAML top level type - *Thing*. *output*s and *effect*s are conditioned using a defined *Condition* object but again their actual values (*ceEffect* and *coOutput*) can be expressed as any kind of DAML object - (*Thing*). It is expected that process definitions extending this top-level ontology will be more specialised by restricting the allowed types. *Condition* has to be a logical formula that evaluates to true or false but the DAML-S specification leaves it for now undefined to serve as a placeholder.

Finally a *Process* class defines a number of timely features: *atTime*, *startTime*, *endTime*, *during*, *timeout*, *timeoutAbsolute*. They are all defined in terms

of *Instant*s or *Interval*s classes from a separate Time ontology (see below).

Figure 10 describes the inners of a *Composite Process*. As such a composite process matches functionally the structure of a Process (above Figure 9 in the sense of *Input*s, *Output*s, *Precondition*s and *Effect*s by having almost equivalents *computedInput*s, *computedOutput*s, *computedPrecondition*s and *computedEffect*s. For now all those are left generic (*Thing*) - so no form of expressing the conditions is specified at this level.

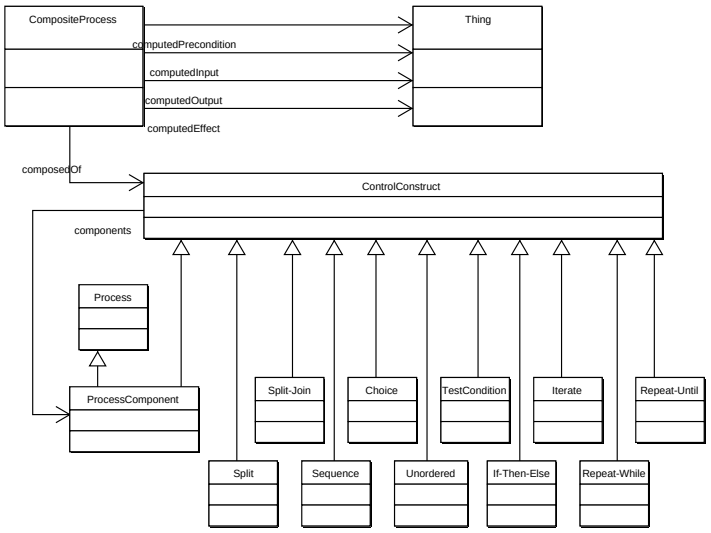

Fig. 10. DAML-S CompositeProcess Definition

The aggregation of sub processes as a *CompositeProcess* is done using *ControlConstruct*s via the *composedOf* property. The Process Ontology defines a number of such constructs like *Sequence*, *Split*, *Split+Join*, *Choice*, *Unordered*, *TestCondition*, *If-Then-Else*, *Iterate*, *RepeatWhile*, *RepeatUntil*. For a full description of those constructs see [ref DAML-S: Semantic] .

In turn a *ControlConstruct* uses a *component* property to indicate the elements forming the construct. It has to be noted that for the *component* property a new class named *ProcessComponent* is introduced which extends both *Process* and *ControlConstruct* so that recursion is possible. In other words a *ControlConstruct* contains either *Process*es or other *ControlConstruct*s.

Each *ControlConstruct* has a different way of specifying the underlying *ProcessComponent*s by specialising the *components* property (E.g. for *Sequence* the

components property holds a *ProcessComponentList* and for *Split* it holds a *ProcessComponentBag*).

Furthermore the DAML-S specification defines a Time ontology which defines two main classes: *Instant*s and *Interval*s. They are reffered by the time properties of the *Process* class.

One of the main features expected in future versions of DAML-S is the specification of the Service Grounding ontologies.

4.3 Web Services Flow Language

The Web Services Flow Language (WSFL) is defined by the specification document ([15]) as:

"an XML language for the description of the composition of Web Services".

WSFL objective is to specifiy:

- usage patterns internal to a Web Service composed from other services. This corresponds to the *FlowModel* defined below.

- external interaction patterns between composed Web Services. This corresponds to the *GlobalModel* defined below.

The core concept of WSFL is the *FlowModel* - a special kind of directed graph. *GlobalModel*s build then on top of *FlowModel*s by defining how they can be brought together. Recursivity is introduced by allowing *FlowModel*s to refer other sub *FlowModel*s.

Central for a *FlowModel* is the concept of *activity* (see Figure 11). A *FlowModel* can define zero or more *activities*. WSFL defines an *activity* as:

"a business task to be performed as a single step within the context of a business process contributing to the overall business goal to be achieved".

Activities correspond to nodes in a graph and they are wired together through *controlLinks*. A flow model can contain zero or more *controlLink*s. A *controlLink* is a directed edge between a *source* activity and a *target* activity. All the *controlLink*s in a *FlowModel* prescribes the order in which activities will have to be performed. The endpoints of the set of all control links that leave a given activity A represent the possible follow-on activities A_1, ..., A_n of activity A.

Which of the activities A_1, ..., A_n actually have to be performed in the concrete instance of the business process (that is,the concrete business context or

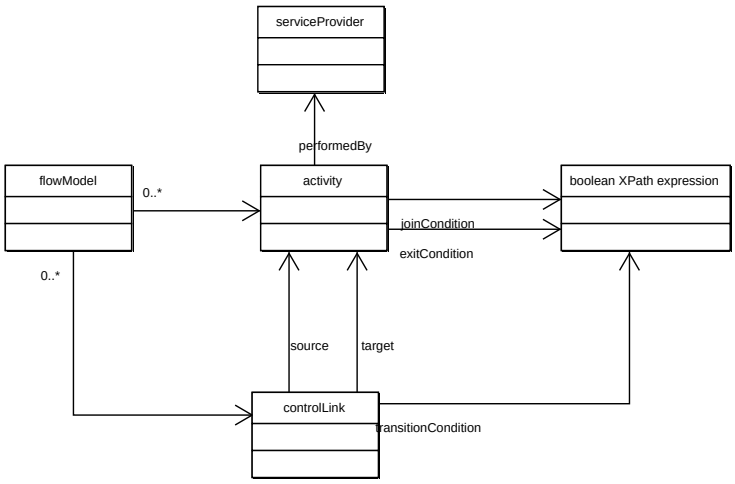

Fig. 11. WSFL - FlowModel internal view

business situation) is determined by so-called *transitionConditions*. Each *controlLink* defines exactly one *transitionCondition* specified as a *boolean expression* (for now defined using the [4] specification). The formal parameters of this expression can refer to messages that have been produced by some of the activities that preceded the source of the control link in the flow.

When an activity A completes, exactly those control links originating at A are followed to their endpoints the transition conditions of which evaluate to true. This set of activities is referred to as *actual follow-on activities* of A in contrast to the full set A_1, ..., A_n of *possible follow-on activities*.

A condition imposed by the WSFL specification is that the *activity/controlLink* graph representation resulting from a *FlowModel* must be *acyclic*. However loops are supported as "do until" constructs.

An *activity* is called a *fork* activity if it has more than one outgoing *controlLink*. In this case all the follow-on activities spawned-off by the *controlLinks* with a true *transitionCondition* will be performed in parallel.

Typically, parallel work has to be synchronised at a later time. Synchronisation is done through *join* activities. An activity is called a *join* activity if it has more than one incoming *controlLink*. By default, the decision whether a join activity is to be performed or not is deferred until all activities that could possibly reach the join are completed. The decision is governed by a *joinCondition* that can be associated with any *activity* and that can be a *boolean expression*.

Sometimes, a weaker semantics of synchronisation is appropriate and supported by the model of WSFL: as soon as the truth-value of a *joinCondition* is known, the associated join activity is dealt with accordingly (that is, either performed or skipped). Control flow that reaches the corresponding *join* activity at a later time is simply ignored.

Activities that have no incoming *controlLink* and that have *joinCondition*s that always evaluate to true are called *start* activities. Activities that have no outgoing *controlLink* are called *end* activities. All *start* activities are automatically scheduled for execution when a *FlowModel* is initially instantiated. When the last *end* activity completes, the output of the overall flow is determined and returned to its invoker.

Each activity has also an associated *exitCondition* which is again a *boolean expression*. The purpose of it is to determine whether or not an activity has to be considered completed. If this is the case the next activities to be performed are determined and execution continues otherwise the activity is executed again.

Because WSFL does not allow for cyclic graphs *exitCondition*s are also used for realizing "do-until" loops. In this case the iteration is done based on the *exitCondition* of an activity which is then implemented by a sub *FlowModel*.

Each activity is *performedBy* a particular *serviceProvider* which defines a set of operations. The operation invoked when an *activity* has to be executed may be perceived as the concrete implementation of the abstract activity.

The *ServiceProvider* concept allows to specify the role that is expected to be played by a potential business partner. Each *ServiceProvider* has a *type* defined by *serviceProviderType*. A *serviceProviderType* is just a named set of *portType*s. In turn a *portType* defines one or more *operation*s. A single *operation* is defined in terms of *input*, *output* and *fault* messages. The defined patterns of interactions for an operation allow for two types of communication: synchronous RPC-like type (for which the request/response, solicit/response patterns are just different perspectives as the party is a client or a server) or a message oriented type (one-way operation from client perspective, notification from a server perspective).

Figure 12 shows the above relations between *FlowModel*s, *ServiceProvider*s, *ServiceProviderType*s and *Opertation*s.

In order to be able to recursively compose *FlowModel*s, WSFL's vision is to perceive them as *serviceProviderType*s (that is, a set of *portType*/*operation*s). For that the WSFL model provides a construct to *export* operations implementing encompassed activities. These exported operations are grouped together to define the public interface of the *FlowModel*. All those operations will be exported that require interactions with some external *ServiceProvider*. These operations

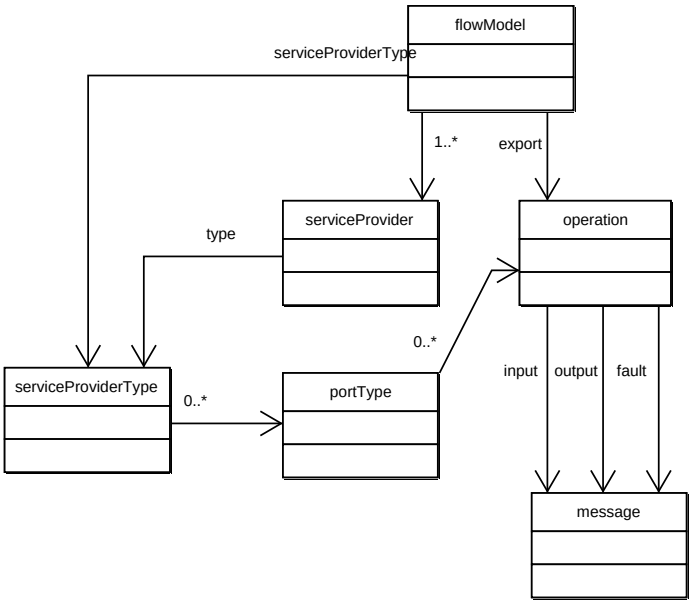

Fig. 12. WSFL - FlowModel external view

specify the interaction offers and demands with respect to the outside world. Stated another way, *activity* implementations that do not require communication with the outside do not need to be exported.

A global model (see Figure 13) defines the interaction between a set of *serviceProvider*s. Interactions are modelled using *PlugLink*s between operations on the *ServiceProviderType*s involved in the composition. Each *PlugLink* is defined between a *source* and a *target operation*. Operations have to be dual, for example, a notification operation on one *ServiceProvider* can be plug linked to a one-way operation on another *ServiceProvider*, or a solicit-response operation can be plug linked to a request-response operation.

Similar[2] to *FlowModel*s, *GlobalModel*s have associated one or more *ServiceProviders* and define themselves for the external world as a *ServiceProviderType*.

Also it has to be noted that the flow model defined by WSFL uses a more reactive approach in contrast with PDDL or DAML-S which take a more classic planning/execution approach.

[2] Actually *FlowModel*s can also have *PlugLink*s and *GlobalModels* can *export* operations but these are rather unusual design choices.

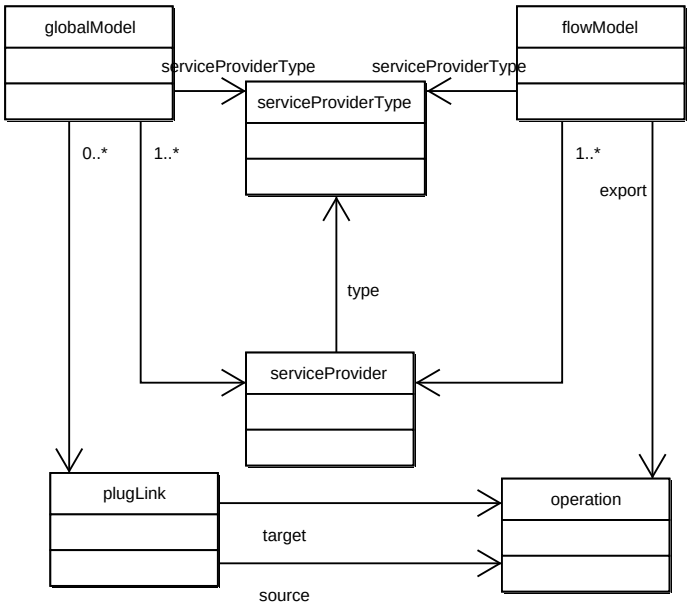

Fig. 13. WSFL - Global Model

4.4 ebXML Business Process Specification Schema

As described on the specification web site ebXML's goal is to:

"provide an open XML-based infrastructure enabling the global use of electronic business information in an interoperable, secure and consistent manner by all parties"

One of the concepts very important for any business is the "Business Process". It describes in detail how Trading Partners take on roles, relationships and responsibilities to facilitate interaction with other Trading Partners in shared collaborations. The interaction between roles takes place as a choreographed set of business transactions. Each business transaction is expressed as an exchange of electronic Business Documents.

The Bussines Process Specification Schema (see [3]) document supports the specification of Business Transactions and the choreography of Business Transactions into Business Collaborations. This is described in a formal document with a DTD and an XML Schema. The current version is 1.01. The specification refers also to a couple of other ebXML specification documents.

Figure 14 describes the core element of the Business Process specification - the *BinaryCollaboration*.

A *BinaryCollaboration* defines a protocol of interaction between two *AuthorizedRoles*. One must be designated the *initiatingRole*, and one the *respondingRole*. Each *BinaryCollaboration* can specify a *preCondition* which is defined as a description of the external state required before the collaboration can commence. Also it can define a *postCondition* which is defined as a description of a state that does not exist before the execution of the collaboration but will exist as a result of the execution. A *BinaryCollaboration* can also specify *beginsWhen* and *endsWhen* conditions which are defined as description of an event external to the collaboration that normally causes the commencement respectively the conclusion of the *BinaryCollaboration*. Finally a *BinaryCollaboration* can specify a *timeToPerform* period within which the collaboration must conclude.

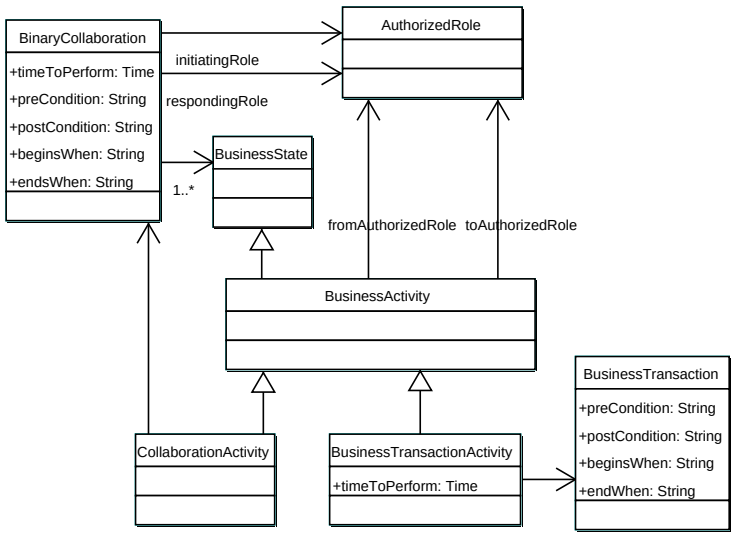

Fig. 14. ebXML - Binary Collaboration

An *AuthorizedRole* represent an actor that is authorised to participate in the collaboration, that is authorised to send the request or response.

A *BinaryCollaboration* consists of one or more *BusinessStates*, some of which are static (e.g. start, fork, end - see below Figure 15), and some of which are action states - the *BusinessActivities*.

A *BusinessActivity* can be either a *BusinessTransactionActivity* or a *CollaborationActivity*, specifying the activity of performing a transaction or another *BinaryCollaboration* respectively.

For any *BusinessActivity* the *fromAuthorizedRole* and *toAuthorizedRole* properties must match one of the *AuthorizedRoles* in the parent *BinaryCollaboration*. The *fromAuthorizedRole* will become the initiator in the *BinaryCollaboration* performed by this activity and the *toAuthorizedRole* will become the responder.

A *CollaborationActivity* is the performance of a *BinaryCollaboration*, possibly within another *BinaryCollaboration*. *BinaryCollaboration*s are re-usable relative to *CollaborationActivity*. The same *BinaryCollaboration* can be performed by multiple *CollaborationActivities* in different *BinaryCollaboration*s, or even by multiple *CollaborationActivities* in the same *BinaryCollaboration*.

A *BusinessTransactionActivity* defines the use of a *BusinessTransaction* within a *BinaryCollaboration*. *BusinessTransaction*s are re-usable relative to *BusinessTransactionActivity*. The same *BusinessTransaction* can be performed by multiple *BusinessTransactionActivities* in different *BinaryCollaborations*, or even by multiple *BusinessTransactionActivities* in the same *BinaryCollaboration*.

A *BusinessTransaction* is an atomic exchange of a set of business information and business signal that must occur in an agreed format, sequence and time period. If any of the agreements are violated then the transaction is terminated and all business information and business signal exchanges must be discarded.

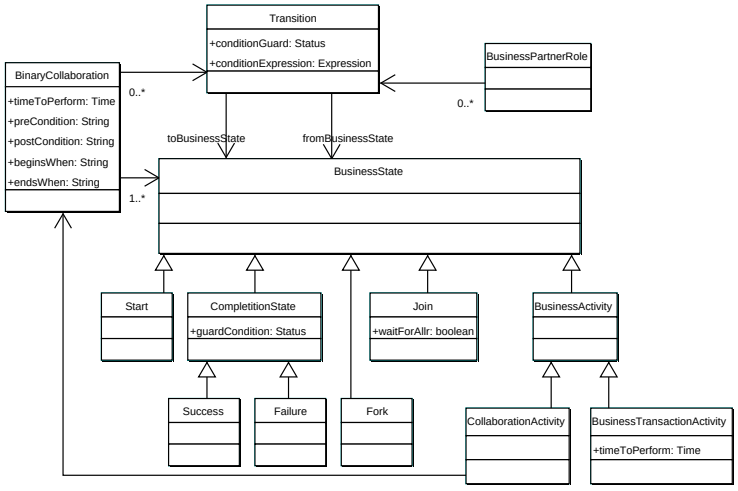

Fig. 15. ebXML - Choregraphy

A Choreography (Figure 15) is an ordering and sequencing of *BusinessActivities* within a Binary Collaboration, or across Binary Collaborations within a *MultipartyCollaboration* (see Figure 16). The choreography is specified in terms of *BusinessState*s, and *Transitions* between those *BusinessState*s.

A *BusinessState* is any state that a *BinaryCollaboration* can be in. A *BusinessActivity* is an abstract kind of *BusinessState* with *BusinessTransactionActivity* and *Collaboration* as concrete subtypes. Also there are a number of auxiliary kinds of *BusinessStates*:

- *Start* - The starting state for a Binary Collaboration. A Binary Collaboration should have at least one starting activity. If none defined, then all activities are considered allowable entry points.
- *CompletionState* - The ending state of an *BinaryCollaboration* which has two subclasses:
  - *Success* - Defines the successful conclusion of a binary collaboration as a transition from an activity.
  - *Failure* - A subtype of CompletionState which defines the unsuccessful conclusion of a binary collaboration as a transition from an activity.
- *Fork* - a state with one inbound transition and multiple outbound transitions. All activities pointed to by the outbound transitions are assumed to happen in parallel.
- *Join* - a state where an activity is waiting for the completion of one or more other activities. Defines the point where previously forked activities join up again.

A *Transition* is between two *BusinessState*s identified trough the properties *fromBusinessState* and *toBusinessState*.

*Transition*s can be gated by *ConditionGuard*s or *ConditionExpression*s which determine whether the *Transition* should happen or not. A *ConditionGuard* refers to the status of the previous transaction and can take one of the fixed values of Success, BusinessFailure, TechnicalFailure or AnyFailure. A *ConditionExpression* can refer to the Document Envelope that caused the transition, the type of Document sent, the content of the document, or postconditions on the prior state.

A *MultipartyCollaboration* is a synthesis of *BinaryCollaboration*s. A *MultipartyCollaboration* consists of two or more of *BusinessPartnerRole*s linked to at most one *AuthorizedRole* in each of the *BinaryCollaboration*s.

The linkage is done via the *Performs* construct and a property for specifying the *AuthorizedRole* which can be either an *initiatingRole* property or a *respondingRole* property.

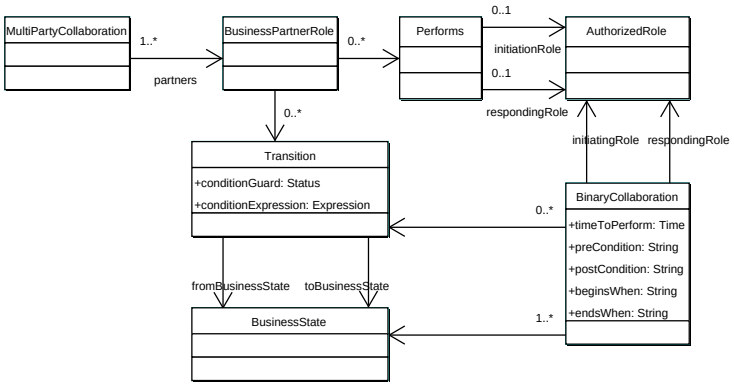

Fig. 16. ebXML - Multiparty Collaboration

Within a *MultipartyCollaboration*, the different *BinaryCollaboration*s can be choreographed by using *Transition*s that create nested *BusinessTransactionActivities*. A nested *BusinessTransactionActivity* is one where a first *Transition* happens after the receipt of the request in the first *Transaction*, and then the entire second *Transaction* is performed before returning to the first *Transaction* to send the response back to the original requestor. In essence an *AuthorizedRole* in one *BinaryCollaboration* receives a request, then turns around and becomes the requestor in an other *BinaryCollaboration* before coming back and sending the response in the first *Binary Collaboration*.

4.5 Concurrent Golog

ConGolog ([11], [13], [11]) is an extended version of the Golog (AlGOl in LOGic, see [14]) language. Golog was originally developed as a high level language for programming robots and software agents. ConGolog is based on a logical formalism, the situation calculus [16], and can model multi-agent processes, nondeterminism and concurrency.

Traditionally simulation was the big virtue of state based formalisms and reasoning over the formal properties of the model was the main advantage of predicative models. By doing a logical definition of an application domain the Golog language can support both simulation and verification.

A domain modelled with ConGolog includes usually two parts:

- A declarative part expressed in **Golog Domain Language** (GDL). GDL describes the domain dynamics, i.e. how the states are modelled, what actions may be performed, when they are possible, what their effects are, and

what is known about the initial state of the system. The GDL declarations can contain:

- Fluents - which describe a situation:
  - relation fluents - relations or properties whose truth value can vary from situation to situation.
  - functional fluents - functions whose value varies from situation to situation.
- Actions - describing what primitive operations can be done. Each action declaration includes a number of preconditions. Actions can be *exogenous* - that is they are performed by actors outside the system.
- Effects - one for each Fluent affected by an Action.

– A procedural part expressed in the **ConGolog Process Description Language** which allows for description of dynamic domains with complex interactions like concurrency and nondeterminism. This can also be viewed as the specification of the behaviour of the agents in the domain. The language includes the following constructs:

- α - a primitive action
- $\phi?$ - wait for a condition
- $(\sigma_1; \sigma_2)$ - sequence
- $(\sigma_1 \mid \sigma_2)$ - nondeterministic choice
- $\pi \bar{x}[\sigma]$ - nondeterministic choice of arguments
- σ^* - nondeterministic iteration
- **if** ϕ **then** σ_1 **else** σ_2 - conditional
- **while** ϕ **do** σ **endWhile** - loop
- $(\sigma_1 \parallel \sigma_2)$ - concurrent execution
- $(\sigma_1 \gg \sigma_2)$ - concurrent execution with different priorities
- $\sigma^{\parallel}$ - concurrent iteration
- $< \bar{x} : \phi \rightarrow \sigma >$ - interrupt
- **proc** $\beta(\bar{x})\sigma$ **endProc** - procedure definition
- $\beta(\bar{t})$ - procedure call
- **noOp** - do nothing

The semantics of both the GDL language and the ConGolog process language are based on *situation calculus*. The frame problem is addressed by allowing for *frame axioms* to be added to the description of the domain and using a *completeness assumption*. The core notion of the framework [3] is the transition, (i.e. a single computation step). A step is either a primitive action or testing whether a condition holds in the current situation. Two special predicates are introduced in order to deal with concurrency:

– *Trans(σ,s,σ',s')* - process σ in situation *s* may legally execute one step, ending in situation *s'* with remaining process σ'.

[3] This is one of the differences between ConGolog and Golog. In Golog the language was directly based on the *Do* predicate.

– *Final(σ,s)* - process σ may legally terminate in situation s.

Then a *Do* predicate (specific to situation calculus) is defined in terms of *Trans* and *Final*.

Finally two main tools are provided for ConGolog :

– a simulation tool for incrementally generating execution traces of ConGolog process specifications. This tool can be used to check whether a model executes as expected in various conditions.
– a verification tool based on logic semantics. With this tool it can be verified that the processes in a domain satisfy certain properties.

4.6 Constraint-Based Agents

Constraint-Based agents is a book describing a new reasoning approach for autonomous agents using local search, CSP and planning techniques.

The planning model is based on Structured CSPs defined also in the book. The planning model roughly includes the following:

– actions - which are considered to consist of a set of preconditions, operations and resulting state changes. They are represented using the following constraints:
  • PreconditionTask - a constraint which tests whenever an action can be carried out.
  • ActionTask - specifies a concrete operation that must be executed within an action.
  • StateTask - describes an action's effect.
  • ActionResource - utilised by an ActionTask for it's execution.
  • StateResource - manages the development of specific properties either in the environment or of the agent itself.

  All the tasks are wired together by suplementar TaskConstraints.
– states - is considered that this is another top-level feature described by PreconditionTasks and StateTasks.
– objects - are defined as constraints aggregated from StateResources .

The model includes also the following top-level features:

– sensors - provide data structured according to objects.
– current time - used for constraint heuristics and cost/goal functions.

Then the model explodes in a variety of relations and constraints between the above concepts without a clear rationale for the design choices made.

5 Examples

Figure 17 shows an example of planning and execution for dynamic service integration. The designer of a text editing application wants to supply a spell-checking functionality. For that it abstracts the behaviour of the spelling system. Then it uses a known interface for spellchecker components which includes structural and behavioural definitions. At runtime the system will then try to broker instances of spellchecking components from the environment. Some of the spell checkers might need in turn a dictionary component which is also abstracted as a dictionary interface. Finally the dictionary might need to use the editor component for interacting with the user regarding ambiguous words.

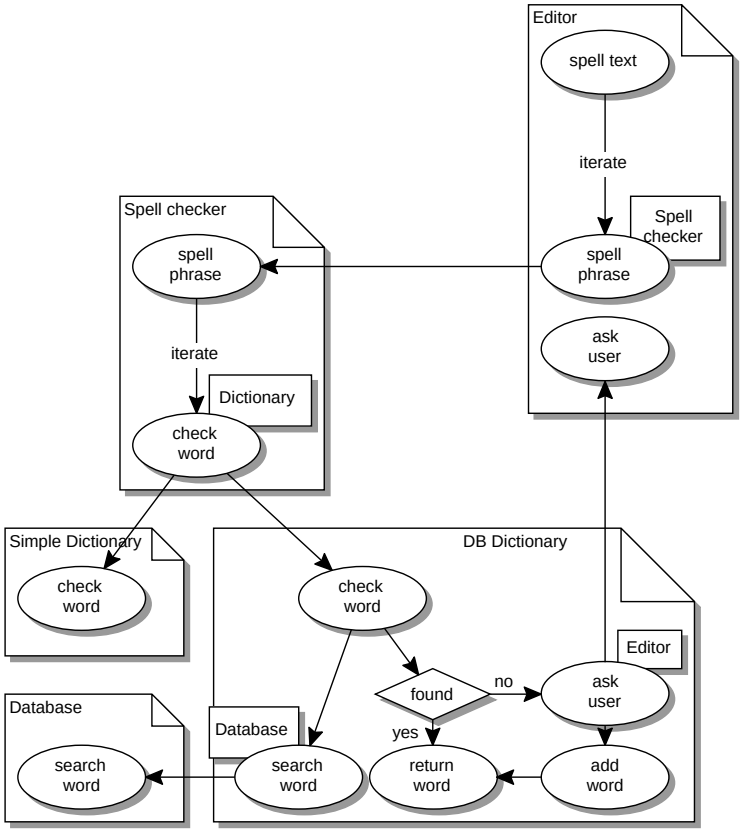

Fig. 17. Sequence of actions needed for spelling.

In Figure 18 we define a possible representation of the Editor/Speller/Dictionary services using DAML-S.

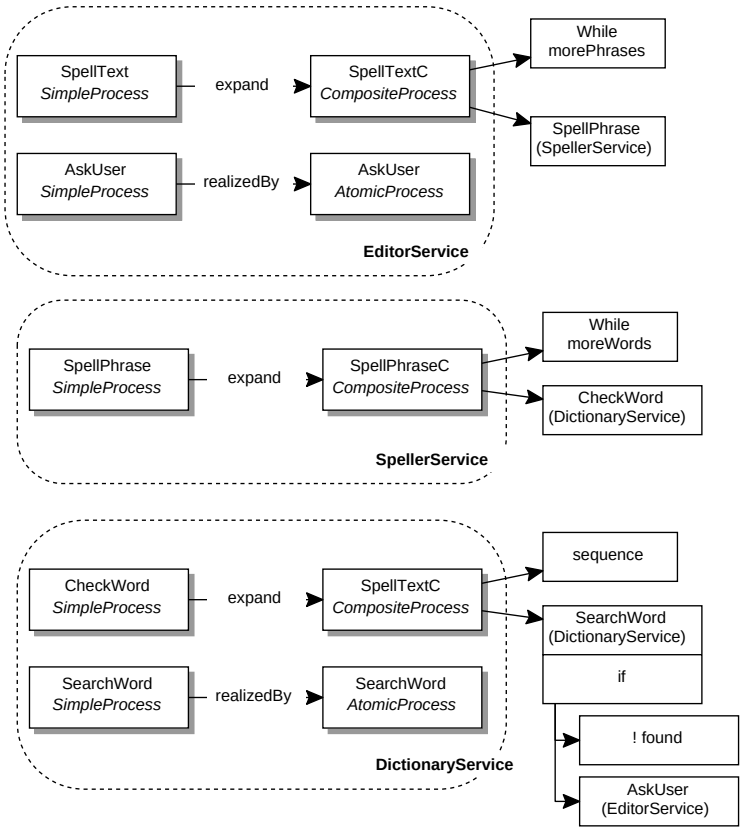

Fig. 18. The Editor/Speller/Dictionary represented as DAML-S specification.

In Figure 19 we define a possible representation of the Editor/Speller/Dictionary services using WSFL.

In Figure 20 we define a possible representation of the Editor/Speller/Dictionary services using ebXML.

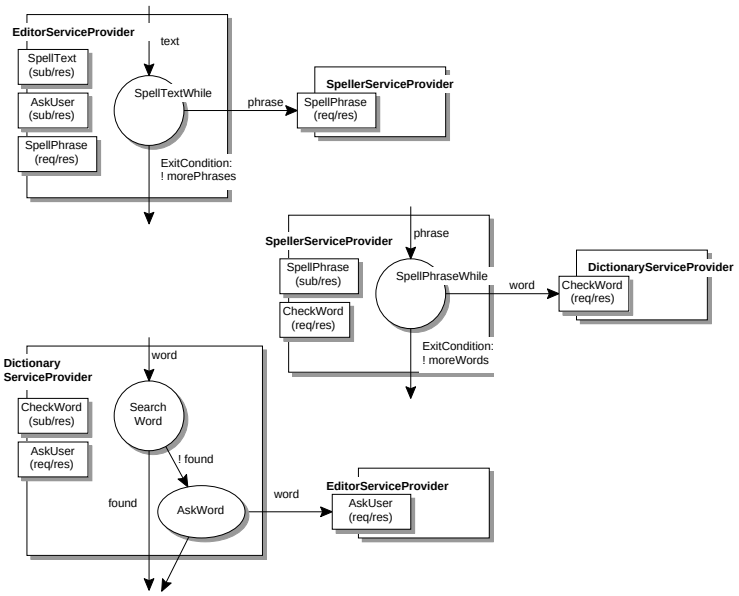

Fig. 19. The Editor/Speller/Dictionary represented as WSFL specification.

6 Comparison of different behavioural formalisms

For comparing the different formalisms presented above we first propose a number of criteria:

- model flexibility and usability - how the model can be actually used
- model reusability and composability - how existing definitions can be reused for defining new definitions and how the model allows definitions to be created by composing other definitions
- model expressiveness - what can be expressed by the model (e.g. preconditions, effects, time, etc)
- model complexity - how complex models can be expressed (e.g. the model allows the specification of states, operations, etc)

6.1 Model Flexibility and Usability

In our analysis we are going to consider the following aspects of the above models:

- grounding (e.g. specifying transport addresses and other deployment parameters).
- binding (e.g. how services can be advertised and searched).
- structural definitions (e.g. structure of possible parameters, signatures of defined operations).

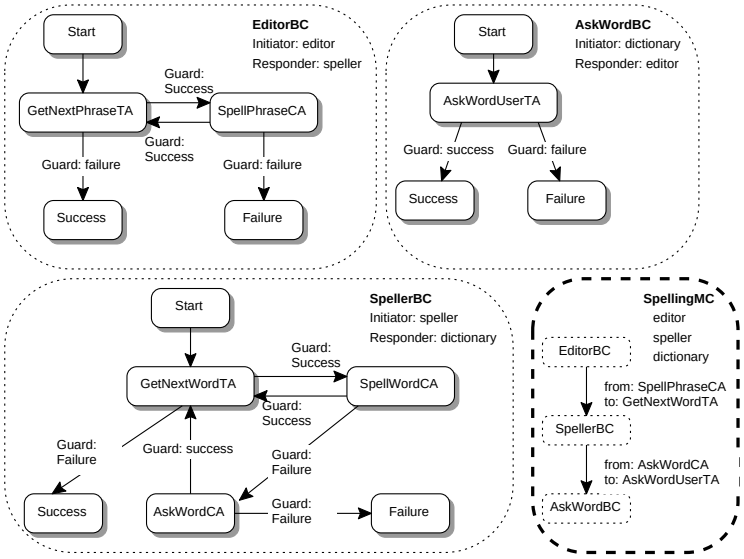

Fig. 20. The Editor/Speller/Dictionary represented as ebXML specification.

– specification of expressions (e.g. transition conditions, preconditions, etc).

Flexibility	PDDL	DAML-S	ebXML	WSFL	ConGolog
Grounding	-	-	Yes	Yes	-
Binding	-	-	Yes	Yes	-
Structural	-	Yes	Yes	Yes	-
Expressions	Yes	-	success/failure guards	Yes	Yes

From the overall flexibility and usability point of view we can consider that WSFL stands the best followed by EbXML. DAML-S, PDDL and ConGolog are more powerful but don't cover all the above aspects.

6.2 Model Reusability and Composability

For defining reusability we considered aspects related to the ability of defining new models using existing ones. This includes also the ability to link to definitions distributed trough the network.

Reusability	PDDL	DAML-S	ebXML	WSFL	ConGolog
Distributed definitions	1.0 only	Yes	Yes	Yes	As multiple prolog files

All DAML-S, ebXML and WSFL provide a very good support for reusability all being XML- based languages. PDDL provides support for multiple file definitions (only 1.0) and finally ConGolog provides support in the measure of the ability of the underlying Prolog interpreter to handle multiple source files.

For composability we considered the structure of the model and the provisions for recursivity. Also we considered the type of relation between a model and it's submodels (e.g. from simple aggregation to complex control flow constructs).

Composability	PDDL	DAML-S	ebXML	WSFL	ConGolog
Recursivity	1.0 only	Yes	Yes	Yes	as procedures
Flow control	1.0 only	Yes	Yes	Yes	Yes

In conclusion the best support for composability is provided by DAMLS-S and ConGolog followed by WSFL. ebXML has a more limited support and the worse can be considered to be PDDL.

6.3 Model Expressiveness

In examining the expressiveness of our models we are going to look at the following features:

- operator parameters
- preconditions (what has to be the state of the world before applying the operator)
- effects (what is the state of the world after applying the operator)
- conditional effects (how the state of the world will change iff a given condition is true after the operator application)
- time related parameters
- distinction between fluents and static variables
- definition of domain axioms or other integrity constraints

Expressiveness	PDDL	DAML-S	ebXML	WSFL	ConGolog
Operator Parameters	Yes	Yes	Yes	Yes	Yes
Preconditions	Yes	Yes	Yes	Yes	Yes
Effects	Yes	Yes	Yes	as exit condition	Yes
Conditional effects	Yes	Yes	-	as exit condition	Yes
Time	Yes	Yes	Yes	-	-
Fluents	Yes	-	-	-	Yes
Axioms	1.0 only	as DAML ontologies	-	-	Yes

We consider that the most expressive language is PDDL allowing for large number of problems to be described. DAML-S is also very promising but unfortunately is still incomplete. ConGolog is also close (in fact DAML-S draws from

both PDDL and ConGolog) in terms of expressiveness. ebXML is more rigid and as such less expressive. Finally we consider WSFL to be the formalism with the worse expressiveness .

6.4 Model Complexity

Since we consider that almost all the presented formalism share the same underlying expressive model we presume that the main difference in the complexity of the expressed instances will come from the complexity of the predicative expressions allowed by the languages.

Complexity	PDDL	DAML-S	ebXML	WSFL	ConGolog
Expressions	first order logic	structured DAML	strings	XPath	Prolog

As such we consider the less complex of all PDDL and close to it Prolog. Since XPath defines a number of internal predicate but allows also for runtime defined predicates we consider it to be more complex. DAML objects specify only the structure of the expressions and as such can be very general. Finally ebXML requires only opaque String expressions or in the case of guardExpressions a String and a language type and as such is the most general and complex case.

6.5 Tradeoffs between expressiveness and complexity

We evaluate PDDL and DAML-S as the best trade-offs between expressiveness and complexity. For PDDL this is due to the fact that the language was very carefully engineered to serve as a testbed in the AIPS [1] competition. For DAML-S this is due to the fact that it was defined as a carefully choose subset of PDDL and ConGolog. Since ebXML has a moderated expressiveness but also strict runtime constraints we consider that also the complexity of possible problems is bounded. WSFL presents the worse trade-off since with a moderated expressiveness allows for very complex models. Even if both ebXML and WSFL are oriented more to be used directly in execution or simulation environments ebXML has still a number of provisions (preconditions, effect) for allowing some kind of off-line reasoning.

Finally as expressiveness and complexity are very tightly linked an evaluation of the trade-off between the two will mainly take into account requirements specific to the application domain.

7 Conclusion

Our evaluation indicates that the reviewed formalisms form roughly three groups:

1. Complete formalisms which have a narrow application scope, have less expressivity and are focused more on runtime-interpretation rather than offline reasoning. This group contains ebXML.
2. High level formalisms which allow a wide range of formalisms to be used within their overall framework - providing wide flexibility but also potentially unbounded complexity. This groups includes DAML-S and WSFL.
3. Formalisms which are more expressive, allow for offline reasoning but are not directly usable in runtime environments. This group contains PDDL and ConGolog.

This provides a spectrum of different solutions - none of which is an obvious choice for system development. In particular we feel that there are a number of interesting research questions which arise from this:

- **Dealing with Heterogeneity**: given that it is unlikely that all future services will be described using the same formalism how do we deal with this resulting heterogeneity? To what extent can mappings be identified between formalisms?
- **Expressivity / Complexity tradeoff**: how well do the current formalisms match the requirements for description of planned services?
- **Combination of formalisms**: for formalisms such as DAML-S which allow arbitrary description components (e.g. languages for preconditions) what are the implications of picking particular formalisms for these tasks (e.g. predicate logic).

References

1. International Planning Competition, http://www.dur.ac.uk/d.p.long/competition.html, 2002.
2. A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. DAML-S (version 0.6) Walk-Through, 2001.
3. Business Process Project Team. ebXML Business Process Specification Schema Version 1.01, May 2001.
4. World Wide Web Consortium. XML Path Language (XPath) Version 1.0, 1999.
5. I. Constantinescu and B. Faltings. Behaviour Description Formalisms for Service Integration: Technical Report No XXX. Technical report, Artificial Intelligence Laboratory, Swiss Federal Institute of Technology, 2002.
6. DAML Services Coalition (alphabetically A. Ankolekar and M. Burstein and J. Hobbs and O. Lassila and D. Martin and S. McIlraith and S. Narayanan and M. Paolucci and T. Payne and K. Sycara and H. Zeng). DAML-S: Semantic Markup for Web Services. In *Proceedings of International Semantic Web Working Symposium (SWWS)*, 2001.
7. DARPA Agent Markup Language Program. Reference description of the DAML+OIL (March 2001) ontology markup language, 2001.
8. M. Fox and D. Long. PDDL+: An extension to PDDL2.1 for modelling planning domains with continuous time-dependent effects, 2002.

9. M. Fox and D. Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains, 2002.
10. M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL–The Planning Domain Definition Language, 1998.
11. Giuseppe De Giacomo, Yves Lesperance, and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.
12. O. Lassila and K. Swick. Resource Description Framework (RDF) model and syntax specification. Technical report, World Wide Web Consortium, 1999.
13. Yves Lesperance, Todd G. Kelley, John Mylopoulos, and Eric S. K. Yu. Modeling dynamic domains with ConGolog. In *Conference on Advanced Information Systems Engineering*, pages 365–380, 1999.
14. Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.
15. F. Leymann. Web Services Flow Language (WSFL 1.0), May 2001.
16. J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, 1969.
17. S. McIlraith, T.C. Son, and H. Zeng. Mobilizing the Semantic Web with DAML-Enabled Web Services. In *Proceedings Second Int'l Workshop Semantic Web (SemWeb'2001)*, 2001.
18. Sheila McIlraith and Tran Cao Son. Adapting golog for programming the semantic web.
19. M. Wooldridge and N. R. Jennings. Towards a Theory of Cooperative Problem Solving. In *Proceedings Workshop Modelling Autonomous Agents in a Multi Agent World (MAAMAW'94)*, pages 15–26. 1994.